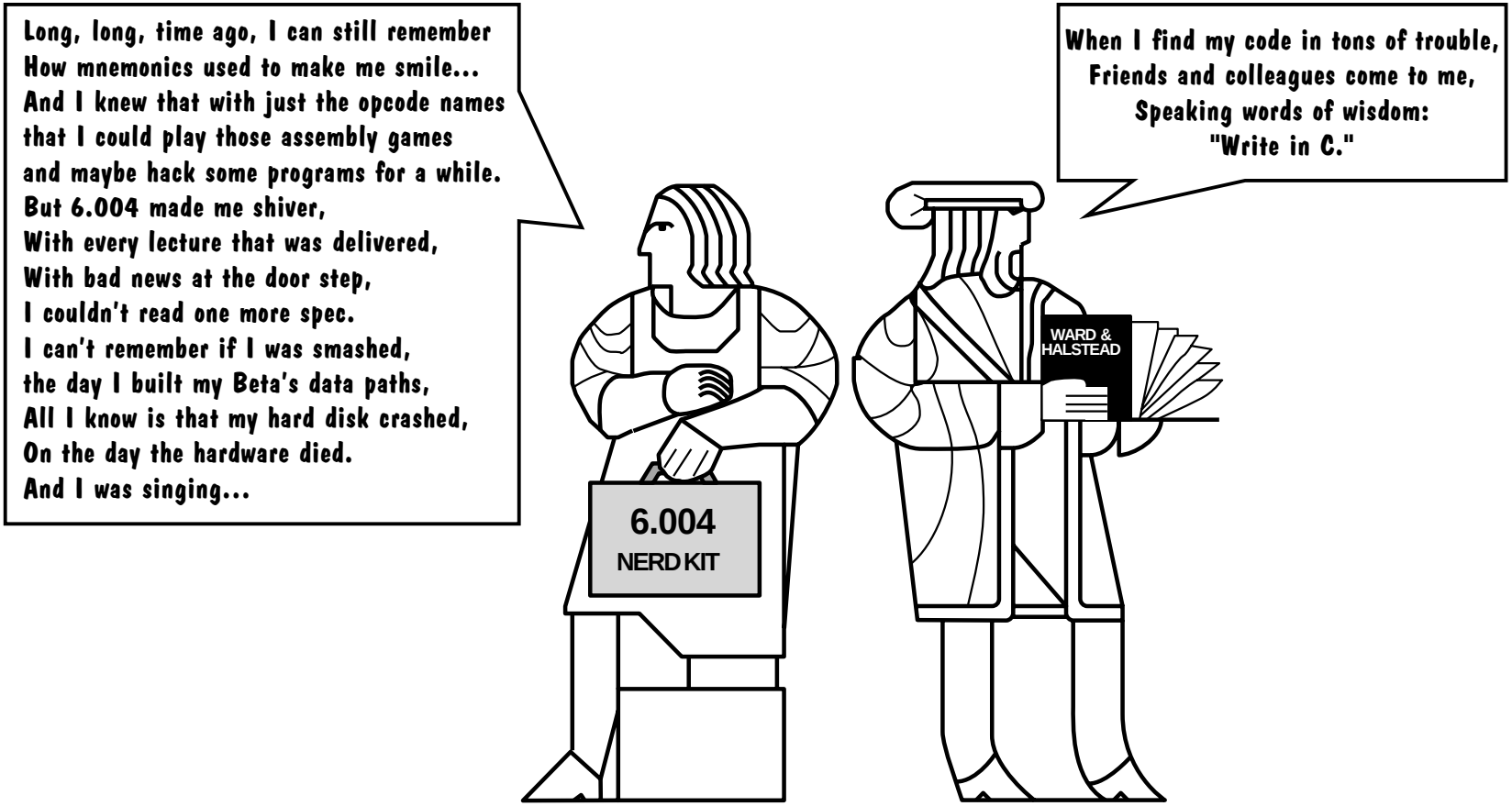
Machine Language, Assemblers, and Compilers



Handouts: Lecture Notes, PS6, Lab 5

Beta Branch Instructions

OPCODE	RC	RA	16-BIT SIGNED CONSTANT

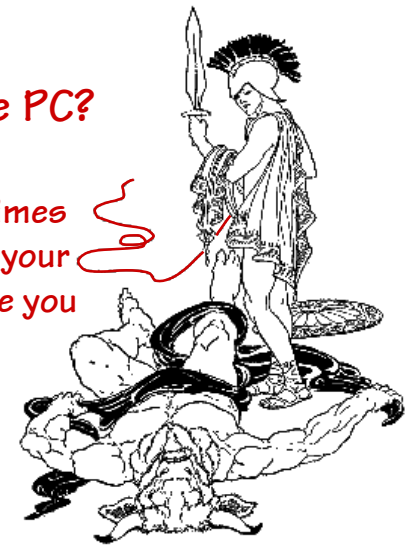
BNE: Branch if not equal

```
PC = PC + 4;
if (REG[RA] != 0) {
   Reg[RA] = PC;
   PC = PC + 4*offset;
} else {
   Reg[RC] = PC;
}
```

Default behavior of the "instruction fetch" state machine

Why save the PC?

Because, sometimes you need to find your way back to where you came from!



BEQ: Branch if equal

```
PC = PC + 4;
if (REG[RA] == 0) {
   Reg[RC] = PC;
   PC = PC + 4*offset;
} else {
   Reg[RC] = PC;
}
```

JMP: Jump

```
temp = PC + 4;
PC = Reg[RA] & ~3;
Reg[RC] = temp;
```

NB: "offset" is a SIGNED CONSTANT encoded as part of the instruction!

Beta Compare Instructions

OPCODE	RC	RA	RB	UNUSED

OPCODE	RC	RA	16-BIT SIGNED CONSTANT

CMPEQ: Test if equal
CMPEQC:

```
if (Reg[RA] == {Reg[RB] or C})
    Reg[RC] = 1;
else
    Reg[RC] = 0;
```

CMPLE: Test if less
CMPLEC: than or equal

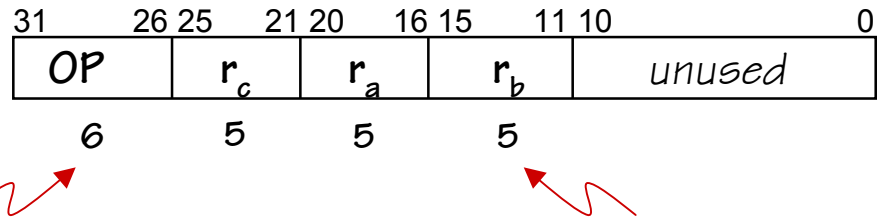
```
if (Reg[RA] <= {Reg[RB] or C})
    Reg[RC] = 1;
else
    Reg[RC] = 0;
```

CMPLT: Test if less
CMPLTC: than

```
if (Reg[RA] < {Reg[RB] or C})
    Reg[RC] = 1;
else
    Reg[RC] = 0;
```

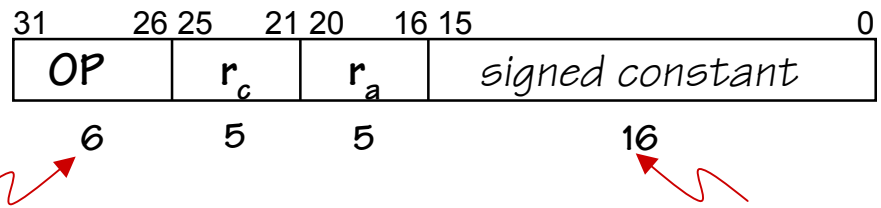
Compare instructions are usually followed by a branch

β Instruction Summary



arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR
shift: SHL, SHR, SAR

Ra and Rb are the operands,
Rc is the destination.
R31 reads as 0, unchanged by writes



arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SARC

Two's complement 16-bit constant for numbers from –32768 to 32767; sign-extended to 32 bits before use.

branch: BNE/BT, BEQ/BF (const = word displacement from PC_{NEXT})
jump: JMP (const not used)
memory access: LD, ST (const = byte offset from Reg[ra])

Binary Instruction Encoding

32-bit (4-byte) ADD instruction:

1 0 0 0 0 0	0 0 1 0 0	0 0 0 1 0	0 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0
OpCode	Rc	Ra	Rb	(unused)

Means, to BETA, *Reg[4] = Reg[2] + Reg[3]*

But, most of us would prefer to write

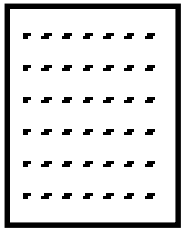
ADD(R2, R3, R4) *(ASSEMBLER)*

or, better yet,

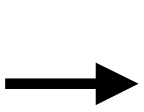
a = b+c; *(C)*

A Program for Writing Programs

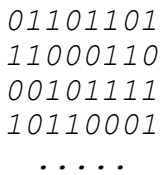
UASM - the 6.004 (Micro) Assembly Language



Symbolic SOURCE text file

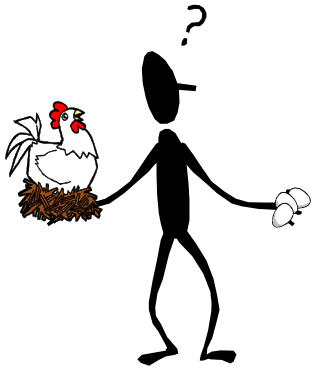


UASM Translator program



Binary Machine Language

STREAM of Bytes to be loaded into memory



UASM:

1. A Symbolic LANGUAGE for representing strings of bits
2. A PROGRAM for translating UASM source to binary.

UASM Source Language

A UASM SOURCE FILE contains, in symbolic text, values of successive bytes to be loaded into memory... e.g. in

`37  -3  255`	decimal (default);
`0b100101`	binary (note the "0b" prefix);
`0x25`	hexadecimal (note the "0x" prefix);

Values can also be expressions; eg, the source file

```
37+0b10-0x10   24-0x1   4*0b110-1   0xF7&0x1F
```

generates 4 bytes of binary output, each with the value 23!

Symbolic Gestures

We can also define SYMBOLS for use in source programs:

```
x = 0x1000          | A variable location
y = 0x1004          | Another variable

| Symbolic names for registers:
R0 = 0
R1 = 1
 ...
R31 = 31
```

Special variable "." (period) means next byte address to be filled:

```
. = 0x100                  | Assemble into 100
   1  2  3  4
five = .                   | Symbol "five" is 0x104
   5  6  7  8
. = .+16                   | Skip 16 bytes
   9 10 11 12
```

Labels (Symbols for Addresses)

LABELS are symbols that represent memory addresses. They can be set with the following special syntax:

`x:` is an abbreviation for "**`x = .`**"

An Example--

```
---- MAIN MEMORY ----
  1000: 09 04 01 00
  1004: 31 24 19 10
  1008: 79 64 51 40
  100c: E1 C4 A9 90
  1010: 00 00 00 10
         3  2  1  0
```

```
. = 0x1000
sqrs: 0 1 4 9
      16 25 36 49
      64 81 100 121
      144 169 196 225
slen: LONG(. - sqrs)
```

Mighty Mighty Macros

Macros are parameterized abbreviations, or shorthand

```
| Macro to generate 4 consecutive bytes:
.macro consec(n)  n  n+1  n+2  n+3

| Invocation of above macro:
consec(37)
```

Has same effect as:

```
37    38    39    40
```

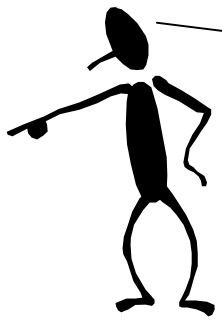
Here are macros for breaking multi-byte data types into byte-sized chunks

```
| Assemble into bytes, little-endian:
.macro WORD(x) x%256 (x/256)%256
.macro LONG(x) WORD(x) WORD(x >> 16)

LONG(0xdeadbeef)
```

Has same effect as:

```
0xef   0xbe   0xad   0xde
```



Boy, that's hard to read. Maybe, those big-endian types do have a point.

Some Handy Bit-Twiddling

OPCODE	RC	RA	RB	UNUSED
110001	00000	01111	10000	000000000000

```
| Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}

| Assemble Beta opc instructions
.macro betaopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
}

| Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL)     betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

".align 4" ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

For Example:

```
Reg[0] = Reg[15]-(-32768) --> betaopc(0x31,15,-32768,0)
```

Finally, Beta Instructions

```
| BETA Instructions:
.macro ADD(RA,RB,RC)   betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)
.macro AND(RA,RB,RC)   betaop(0x28,RA,RB,RC)
.macro ANDC(RA,C,RC)   betaopc(0x38,RA,C,RC)
.macro MUL(RA,RB,RC)   betaop(0x22,RA,RB,RC)
.macro MULC(RA,C,RC)   betaopc(0x32,RA,C,RC)
   .
   .
   .
.macro LD(RA,CC,RC)    betaopc(0x18,RA,CC,RC)
.macro LD(CC,RC)       betaopc(0x18,R31,CC,RC)
.macro ST(RC,CC,RA)    betaopc(0x19,RA,CC,RC)
.macro ST(RC,CC)       betaopc(0x19,R31,CC,RC)
   .
   .
   .
.macro BEQ(RA,LABEL,RC)       betabr(0x1D,RA,RC,LABEL)
.macro BEQ(RA,LABEL)          betabr(0x1D,RA,r31,LABEL)
.macro BNE(RA,LABEL,RC)       betabr(0x1E,RA,RC,LABEL)
.macro BNE(RA,LABEL)          betabr(0x1E,RA,r31,LABEL)
```

(See beta.uasm)

Example Assembly

```
ADDC(R3,1234,R17)
```

expand ADDC macro with RA=R3, C=1234, RC=R17

```
betaopc(0x30,R3,1234,R17)
```

expand betaopc macro with OP=0x30, RA=R3, CC=1234, RC=R17

```
.align 4
LONG((0x30<<26)+((R17%32)<<21)+((R3%32)<<16)+(1234 % 0x10000))
```

expand LONG macro with X=0xC22304D2

```
WORD(0xC22304D2)    WORD(0xC22304D2 >> 16)
```

expand first WORD macro with X=0xC22304D2

```
0xC22304D2%256    (0xC22304D2/256)%256    WORD(0xC223)
```

evaluate expressions, expand second WORD macro with X=0xC223

```
0xD2    0x04    0xC223%256    (0xC223/256)%256
```

evaluate expressions

```
0xD2    0x04    0x23    0xC2
```

Don't have it? Fake it!

Convenience macros can be used to extend our assembly language!

```
.macro MOVE(RA,RC)        ADD(RA,R31,RC)    | Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)       ADDC(R31,C,RC)    | Reg[RC] <- C

.macro COM(RA,RC)         XORC(RA,-1,RC)    | Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)         SUB(R31,RB,RC)    | Reg[RC] <- -Reg[RB]
.macro NOP()              ADD(R31,R31,R31)  | do nothing

.macro BR(LABEL)          BEQ(R31,LABEL)    | always branch
.macro BR(LABEL,RC)       BEQ(R31,LABEL,RC) | always branch
.macro CALL(LABEL)        BEQ(R31,LABEL,LP) | call subroutine
.macro BF(RA,LABEL,RC)    BEQ(RA,LABEL,RC)  | 0 is false
.macro BF(RA,LABEL)       BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)    BNE(RA,LABEL,RC)  | 1 is true
.macro BT(RA,LABEL)       BNE(RA,LABEL)

| Multi-instruction sequences
.macro PUSH(RA)           ADDC(SP,4,SP)   ST(RA,-4,SP)
.macro POP(RA)            LD(SP,-4,RA)    ADDC(SP,-4,SP)
```

Our First β Program

```
.include beta          | Load macros, etc
       LD(input, r0)   | Set up r0,
       BR(bitrev)      |   then do it!
input: LONG(0x12345)   | Input (in HEX)

| Reverse the bits in r0, leave result in r1
bitrev:
       CMOVE(32,r2)    | 32-bit counter.
       CMOVE(0,r1)     | clear, for output
loop:  ANDC(r0,1,r3)   | Low-order bit
       SHLC(r1,1,r1)   | shift it into
       OR(r3,r1,r1)    |   output word
       SHRC(r0,1,r0)   | strip input bit
       SUBC(r2,1,r2)   | count bits...
       BNE(r2,loop)    | ...until done.
       HALT()          | Stop program!
```

The Result...

(using bsim "batch" mode)

```
>uasm bitrev
>bsim -b bitrev
File bitrev.bin loaded; 8192 DRAM bytes initialized.
stopped...bitrev halted at <PC> = 0x80000048. Registers:
  r0: 0x00000000 0xa2c48000 0x00000000 0x00000000
  r4: 0x00000000 0x00000000 0x00000000 0x00000000
  r8: 0x00000000 0x00000000 0x00000000 0x00000000
 r12: 0x00000000 0x00000000 0x00000000 0x00000000
 r16: 0x00000000 0x00000000 0x00000000 0x00000000
 r20: 0x00000000 0x00000000 0x00000000 0x00000000
 r24: 0x00000000 0x00000000 0x00000000 0x00000000
 r28: 0x80000010 0x00008000 0x00000000 0x00000000

>
```

β

Interactive Simulation

```
6.004 BETA Simulator (h for help)                                  X=- dcbavrcsix
Instrs: 00000027
------ Machine Registers ------    ---- MAIN MEMORY ----    ------ STACK ------
R00: 0000048D     R16: 00000000         =                   7fd0:   00 00 00 00
R01: 00000028     R17: 00000000                             7fd4:   00 00 00 00
R02: 0000001A     R18: 00000000                             7fd8:   00 00 00 00
R03: 00000000     R19: 00000000                             7fdc:   00 00 00 00
R04: 00000000     R20: 00000000                             7fe0:   00 00 00 00
R05: 00000000     R21: 00000000                             7fe4:   00 00 00 00
R06: 00000000     R22: 00000000                             7fe8:   00 00 00 00
R07: 00000000     R23: 00000000                             7fec:   00 00 00 00
R08: 00000000     R24: 00000000                             7ff0:   00 00 00 00
R09: 00000000     R25: 00000000                             7ff4:   00 00 00 00
R10: 00000000     R26: 00000000                             7ff8:   00 00 00 00
R11: 00000000      BP: 00000000                             7ffc:   00 00 00 00
R12: 00000000      LP: 00000000                             8000:S--00 00 00 00
R13: 00000000      SP: 00008000                             8004:   00 00 00 00
R14: 00000000      XP: 00000000                             8008:   00 00 00 00
R15: 00000000      PC: 80000024                             800c:   00 00 00 00

        BNE(r2,loop)            | ...until done.
```

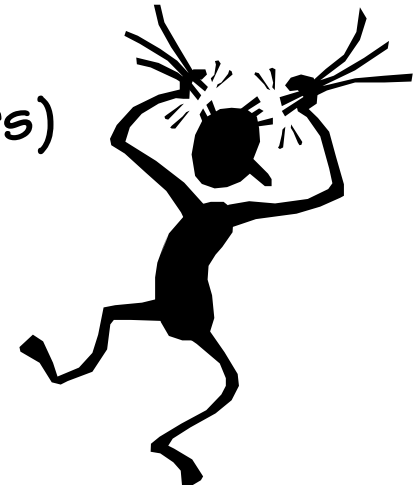
Arghh, this is Painful!

Crazy acronyms are only significantly better than bits.

We need a higher level language!

Even with an assembler there is still a lot of low-level book keeping to be concerned with:

- Resource allocation
  (Where variables are in memory or registers)
- Code placement and organization
- Documentation and readability
- Portability



High-level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm for computing a factorial:

```
int bitrev(int n)
{
   int count=32;
   int result = 0;

   do {
      result = (result << 1) | (1 & n);
      n = n>>1;
      count = count-1;
   } while (n != 0);
   return result;
}
```

Here we've expressed our bit reverse algorithm using the high-level semantics of the "C programming language."

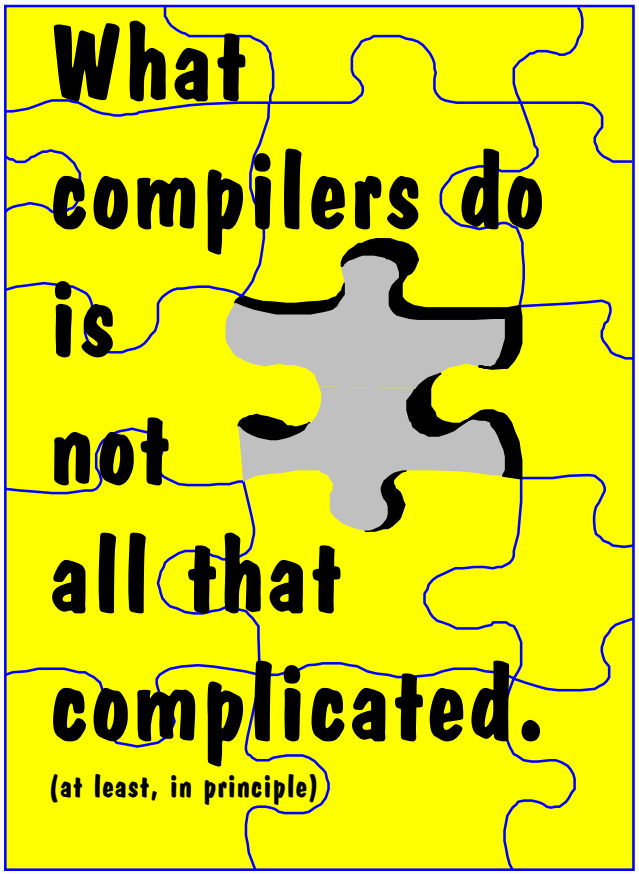
Why?

- more readable
  (by both human and machine)
- concise
- unambiguous
- portable
  (algorithms frequently outlast their HW platforms)

How Compilers Work

At the highest level, compilers match simply search for special keywords and replace them with assembly language templates.

The generation of modern compilers is highly automated. This automation is one of the great successes of computer science. Many portable compilers only require that a "backend" of code fragments or "templates" be generated in the assembly language of the target machine.



Compiling Expressions

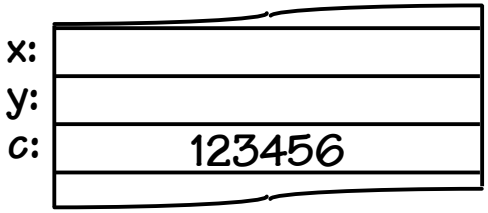
C code:

```
int x, y;
y = (x-3)*(y+123456)
```

Beta assembly code:

```
x:      LONG(0)
y:      LONG(0)
c:      LONG(123456)
...

        LD(x, r1)
        SUBC(r1,3,r1)
        LD(y, r2)
        LD(C, r3)
        ADD(r2,r3,r2)
        MUL(r2,r1,r1)
        ST(r1,y)
```



- VARIABLES are assigned memory locations and accessed via LD or ST
- OPERATORS translate to ALU instructions
- SMALL CONSTANTS translate to "literal-mode" ALU instructions
- LARGE CONSTANTS translate to initialized variables

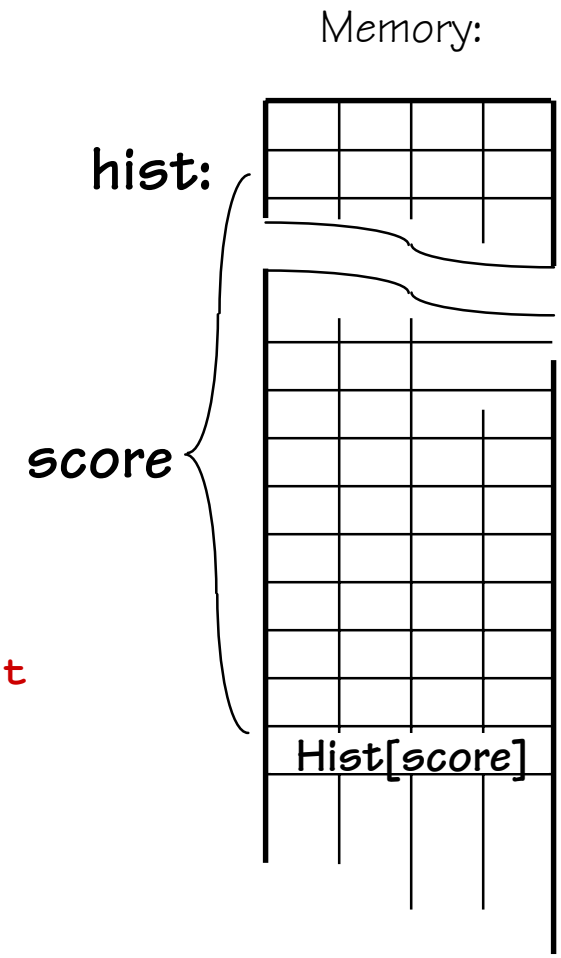
Data Structures: Arrays

The C source code

```
int Hist[100];
...
Hist[score] += 1;
```

might translate to:

```
hist:   .=.+4*100
...
<score in r1>
MULC(r1,4,r2)     | index -> byte offset
LD(r2,hist,r0)    | hist[score]
ADDC(r0,1,r0)     | increment
ST(r0,hist,r2)    | hist[score]
```



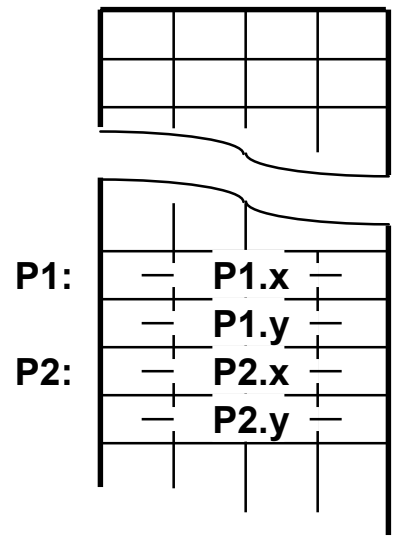
Address:

CONSTANT base address +
VARIABLE offset computed from index

Data Structures: Structs

```
struct Point {
   int x, y;
} P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 157;
```

Memory:



might translate to:

```
P1:  .=.+8
P2:  .=.+8
x=0            | Offset for x component
y=4            | Offset for y component
...
ADDC(r31,157,r0)     | r0 <- 157
ST(r0,P1+x)          | P1.x = 157
...
<p in r3>
ST(r0,y,r3)          | p->y = 157;
```

Address:

VARIABLE base address +
CONSTANT component offset

Conditionals

C code:

```
if (expr) {
    STUFF
}
```

Beta assembly:

```
    (compile expr into rx)
    BF(rx, Lendif)
    (compile STUFF)
Lendif:
```

C code:

```
if (expr) {
    STUFF1
} else {
    STUFF2
}
```

Beta assembly:

```
    (compile expr into rx)
    BF(rx, Lelse)
    (compile STUFF1)
    BR(Lendif)
Lelse:
    (compile STUFF2)
Lendif:
```

There are little tricks that come into play when compiling conditional code blocks. For instance, the statement:

```
if (y > 32) {
    x = x + 1;
}
```

compiles to:

```
  LD(y,R1)
  CMPLEC(R1,32,R1)
  BT(R1,Lendif)
  ADDC(R2,1,R2)
Lendif:
```

Loops

C code:

```
while (expr) {
    STUFF
}
```

Beta assembly:

```
Lwhile:
    (compile expr into rx)
    BF(rx,Lendwhile)
    (compile STUFF)
    BR(Lwhile)
Lendwhile:
```

Alternate Beta assembly:

```
    BR(Ltest)
Lwhile:
    (compile STUFF)
Ltest:
    (compile expr into rx)
    BT(rx,Lwhile)
Lendwhile:
```

Compilers spend a lot of time optimizing in and around loops.

- moving all possible computations outside of loops
- unrolling loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"

Our Favorite Program

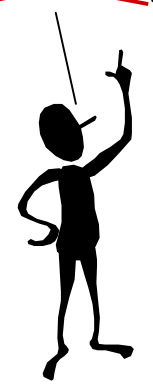
```
int n = 20, r;        n:   LONG(20)
                      r:   LONG(0)

r = 1;                     ADDC(r31, 1, r0)
                           ST(r0, r)
                      loop:
while (n > 0) {            LD(n, r1)
                           CMPLT(r31, r1, r2)
                           BF(r2, done)
                           LD(r, r3)
     r = r*n;              LD(n,r1)
                           MUL(r1, r3, r3)
                           ST(r3, r)
     n = n-1;              LD(n,r1)
                           SUBC(r1, 1, r1)
                           ST(r1, n)
 }                         BR(loop)
                      done:
```

Cleverness:
None…
straightforward compilation

(11 instructions in loop…)

Optimizations are what make compilers complicated ~~interesting~~!



Optimizations

```
int n = 20, r;
r = 1;
while (n > 0) {
    r = r*n;
    n = n-1;
}
```

```
n:    LONG(20)
r:    LONG(0)

      ADDC(r31, 1, r0)
      ST(r0, r)
      LD(n,r1)        ; keep n in r1
      LD(r,r3)        ; keep r in r3
loop:
      CMPLT(r31, r1, r2)
      BF(r2, done)
      MUL(r1, r3, r3)
      SUBC(r1, 1, r1)
      BR(loop)
done:
      ST(r1,n)        ; save final n
      ST(r3,r)        ; save final r
```

Cleverness:
We move LDs/STs out of loop!

(Still, 5 instructions in loop...)

Really Optimizing...

```
int n = 20, r;
r = 1;
while (n > 0) {
    r = r*n;
    n = n-1;
}
```

```
n:    LONG(20)
r:    LONG(0)

      LD(n,r1)            ; keep n in r1
      ADDC(r31, 1, r3)    ; keep r in r3
      BEQ(r1,done)        ; why?
loop:
      MUL(r1, r3, r3)
      SUBC(r1, 1, r1)
      BNE(r1,loop)
done:
      ST(r1,n)            ; save final n
      ST(r3,r)            ; save final r
```

Cleverness:
We avoid overhead of conditional!

(Now 3 instructions in loop...)

$20! = 2{,}432{,}902{,}008{,}176{,}640{,}000 > 2^{61}$

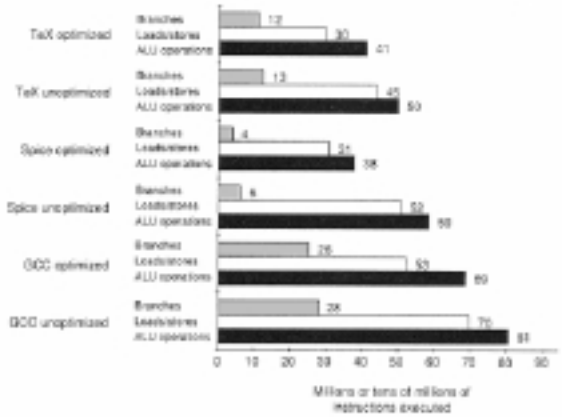
$12! = 479{,}001{,}600 = 0x1c8cfc00$

Optimization Summary

Major classes of optimizations:

Optimization name	Explanation
High-Level	**At or near source level; machine independent**
Procedure integration	Replace procedure call by procedure body (in-lining)
Local	**Within straight-line code**
Eliminate common subexpressions	Replace two instances of the same computation with a single copy
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation
Global	**Across a branch**
Global common subexpressions	Same as local, but this version crosses branches
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., A=X) with X
Code motion	Remove code from a loop that computes the same value each iteration of the loop
Induction variable elimination	Simplify/eliminate array-addressing calculations within loops
Machine-dependent	**Depends on machine knowledge**
Strength reduction	Many examples, such as replace multiply by constant with adds and shifts
Pipeline scheduling	Reorder instructions to avoid pipeline stalls
Branch offset optimization	Choose the shortest branch displacement that reaches target

The effects of optimization in absolute instruction counts



COMING ATTRACTIONS: (AFTER THE BREAK)

Procedures and Stacks